

RL-TR-96-165
Final Technical Report
October 1996



DISTRIBUTED COMPUTING OVER NEW TECHNOLOGY NETWORKS: QUALITY OF SERVICE FOR CORBA OBJECTS

BBN Systems and Technologies

**Dr. David E. Bakken, Dr. Richard E. Schantz,
and Dr. John A. Zinky**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

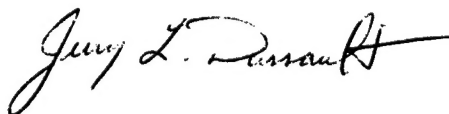
19961125 024

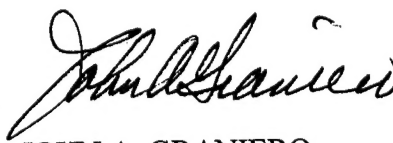
DTIC QUALITY INSPECTED 3

**Rome Laboratory
Air Force Materiel Command
Rome, New York**

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-96-165 has been reviewed and is approved for publication.

APPROVED: 
JERRY L. DUSSAULT
Project Engineer

FOR THE COMMANDER: 
JOHN A. GRANIERO
Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3AB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE October 1996		3. REPORT TYPE AND DATES COVERED Final Jul 94 - May 96
4. TITLE AND SUBTITLE DISTRIBUTED COMPUTING OVER NEW TECHNOLOGY NETWORKS: QUALITY OF SERVICE FOR CORBA OBJECTS			5. FUNDING NUMBERS C - F30602-94-C-0188 PE - 62702F PR - 5581 TA - 28 WU - 33	
6. AUTHOR(S) Dr. David E. Bakken, Dr. Richard E. Schantz, and Dr. John Zinky				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) BBN Systems and Technologies 10 Moulton Street Cambridge, MA 02138			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3AB 525 Brooks Road Rome, NY 13441-4505			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-96-165	
11. SUPPLEMENTARY NOTES RL Project Engineer: Jerry L. Dussault, C3AB, (315) 330-2067				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This project investigated how "middleware" can be extended to allow distributed programs to exploit recent and anticipated advances in networking technology. This was accomplished in four (largely sequential) steps: (1) Study the impact of new technology networks on distributed computing environments, which lead to the conclusion that Quality of Service (QoS) was the most promising attribute of new networking technologies in terms of improving distributed programs such as C3 or collaborative planning applications; (2) Study how Distributed Computing Environments (DCEs) should support QoS in WAN environments; (3) Architecting a QoS Framework for CORBA; and (4) Experimentation. This report describes an architecture for Quality of Service for CORBA Objects (QuO), which was developed to overcome performance limitations encountered in wide-area network environments. Distributed applications must become adaptable to cope with system properties which vary greatly over time, if they are to be acceptable for mission critical use and cost effectively evolve. QuO develops a cohesive framework for constructing adaptable applications by introducing the concepts of quality of service for object access, and by providing the mechanisms which can be used to integrate these concepts into emerging applications.				
14. SUBJECT TERMS Distributed Computing, Quality of Service (QoS), CORBA			15. NUMBER OF PAGES 52	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED		18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED
			20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

DTIC QUALITY INSPECTED 3

Table of Contents

LIST OF FIGURES	iii
GLOSSARY	iv
1. PROJECT SUMMARY	1
1.1 STUDYING IMPACT ON DISTRIBUTED COMPUTING ENVIRONMENTS (DCEs) OF NEW TECHNOLOGY NETWORKS (NTNS).....	1
1.2 STUDYING HOW DCEs SHOULD SUPPORT QUALITY OF SERVICE (QoS) IN WAN ENVIRONMENTS (SOW 4.1.1 & 4.1.2).....	2
1.3 ARCHITECTING QUO: A QoS FRAMEWORK FOR CORBA (SOW 4.1.3).....	3
1.4 EXPERIMENTATION WITH QUO AND COTS (SOW 4.1.4)	3
2. TECHNICAL OVERVIEW	4
2.1 INTRODUCTION	4
2.2 OBSERVATIONS ABOUT DISTRIBUTED APPLICATIONS	5
2.2.1 <i>Overview of Collaborative Planning Applications</i>	6
2.2.2 <i>Problems in Developing and Deploying Distributed Applications</i>	8
2.2.3 <i>QuO Solution</i>	9
2.3 INTEGRATING KNOWLEDGE OF SYSTEM PROPERTIES	10
2.3.1 <i>Connections</i>	11
2.3.2 <i>QoS Regions</i>	12
2.3.3 <i>Adaptivity Implies Multiple Behaviors</i>	14
2.3.4 <i>Summary: QuO Commitment Epochs</i>	15
2.4 REDUCING THE VARIANCE IN SYSTEM PROPERTIES	16
2.4.1 <i>Masking Variance with Layers of Delegates</i>	16
2.4.2 <i>Integrating System Knowledge from Different Sources</i>	18
2.4.3 <i>System Conditions as First Class Objects</i>	20
2.4.4 <i>Summary</i>	22
2.5 OPEN IMPLEMENTATION TECHNIQUES AND QUO OBJECTS	22
2.6 IMPLEMENTATION	24
2.6.1 <i>C++ Objects to Implement the CDL</i>	24
2.7 RELATED WORK	25
2.8 REFERENCES	26
3. CONCLUSIONS AND RECOMMENDATIONS	28
3.1 CORBA AS A WIDE-AREA RESEARCH AND DEVELOPMENT VEHICLE	28
3.2 DCNTN SUMMARY	29
3.2.1 <i>Technical Conclusions</i>	30
3.2.2 <i>QuO Summary</i>	30
3.3 LIMITATIONS OF THE QUO APPROACH	30
3.4 FUTURE DIRECTIONS	31
APPENDIX A: PUBLICATIONS PRESENTATIONS, AND OTHER PARTICIPATION	34
APPENDIX B: CONTRACT DESCRIPTION LANGUAGE (CDL) FOR SCREENSAVER CONTRACT	36

List of Figures

FIGURE 1: COLLABORATIVE PLANNING USAGE PATTERNS	7
FIGURE 2: SCREENSAVER REGIONS	13
FIGURE 3: MASKING SYSTEM PROPERTIES.....	18
FIGURE 4: DELEGATE OBJECT LAYERS	20
FIGURE 5: AGGREGATION VIA A SHARED SYSTEM CONDITION OBJECT	21
FIGURE 6: A METAOBJECT PROTOCOL	23
FIGURE 7: C++ OBJECTS WHICH IMPLEMENT THE SCREENSAVER CONTRACT	25
FIGURE 8: MIDDLEWARE AND RESERVATIONS RESOURCES	28
FIGURE 9: QUO IMPLEMENTATION STATUS AND FUTURE DIRECTIONS.....	32

Glossary

COTS Commercial Off-The-Shelf. A product available commercially.

CORBA Common Object Request Broker Architecture. The OMG's distributed object standard; see OMG below.

DCNTN Distributed Computing over New Technology Networks. This project.

GOTS Government Off-The-Shelf. Software or other technology owned by the government.

MOP MetaObject Protocol. Allows the client to provide an object to choose the implementation for a module which best suits its needs.

OI Open Implementation. Discipline that allows the designer of an object to expose key design decisions which affect the object's performance and reliability.

OMG Object Management Group. The largest consortium in the world, with over 500 software and hardware vendors, end users, and government agencies. The OMG is developing the CORBA standard. See URL <http://www.omg.org>.

QDL QoS Description Language. Specifies an applications expected usage patterns and QoS requirements for a connection to an object.

QoS Quality of Service.

QuO Quality of Service for CORBA Objects, the framework developed as part of the DCNTN project.

WAN Wide-Area Network.

1. Project Summary

This project investigated how middleware can be extended to allow distributed programs to exploit recent and anticipated advances in networking technology, as outlined in the SOW:

OBJECTIVE: The objective of this effort is to explore the impact of new technology networks and communication services on distributed computing environments.

SCOPE: The scope of this effort shall include an investigation of trends in both networking technology and Distributed Computing Environment (DCE) technology. The impact of technology trends shall be assessed, and experiments shall be conducted to test the effects of new communication services on emerging DCE architectures, distributed information environments, and applications.

We did this in four (largely sequential) steps:

1. Study the impact of new technology networks on distributed computing environments, which lead to the conclusion that QoS was the most promising attribute of new networking technologies in terms of improving distributed programs such as C3 or collaborative planning applications.
2. Study how DCEs should support QoS in WAN environments.
3. Architecting a QoS Framework for CORBA.
4. Experimentation with QuO and COTS.

These steps are now examined in turn. A list of publications, presentations, and other participation which resulted from this effort (which are cited in the first four subsections) appears in Appendix A.

1.1 Studying Impact on Distributed Computing Environments (DCEs) of New Technology Networks (NTNs)

SOW 3.3 summarizes the aims of this step:

To best take advantage of the facilities offered by these emerging network technologies, the opportunities they furnish for implementing distributed information environments and their impact on DCEs must be carefully studied.

After careful study, our conclusion was that the two most promising networking technologies in terms of their potential impact on WAN applications were:

- Switch support for multicast
- Quality of Service (QoS)

Since many researchers and companies were working on multicast, and none on QoS for WANs, we focused the rest of this project on the issues involved with providing QoS for WAN applications.

This step resulted in presentations PRES-01, PRES-02, and PRES-03.

1.2 Studying How DCEs Should Support Quality of Service (QoS) in WAN Environments (SOW 4.1.1 & 4.1.2)

The WAN Environment is considerably more difficult to develop and maintain distributed applications for than is a LAN, because

- The system properties (non-functional issues such as performance, failures, etc) which must be dealt with are much more variable and dynamic during a given execution of a program.
- The system properties are much more likely to vary from configuration to configuration.

As a result of this, our conclusions as to how QoS should be supported to be useful across a WAN are:

1. CORBA is an excellent vehicle for wide-area distributed R&D. Distributed object technology encapsulates the three kinds of resources which operating systems typically encapsulate — processing, storage, and communications — and is thus quite general. CORBA is the best distributed technology available, with its technology arguably 3 years ahead of that of network OLE. The research issues involved in supporting CORBA's functional interoperability have largely been resolved, with only engineering issues remaining to be resolved. See Section 3.1 for a further discussion of this conclusion.
2. To be most useful in the wide area, an object's functional interface (e.g., CORBA IDL) must be augmented to deal with non-functional aspects such as delay and throughput. Bad performance cannot be hidden beneath a functional interface: it will always show through.
3. Client-object contracts which span wide areas *will* be broken, because WAN environments are so dynamic and hostile. To deal with this, we must have compound contracts with fallback positions which are specified and automatically deployed when necessary.
4. Adaptivity implies multiple implementations. A given implementation of an object makes design tradeoffs *viz* processing versus storage versus communications bandwidth. If an application is to be successfully fielded in multiple environments, including WANS, a number of implementations with different tradeoffs should be available and deployed as conditions merit. A client should be able to access one implementation if the object is local to its desktop, another implementation if it is

across a WAN, and yet another if the bandwidth across that WAN is unusually low or high.

5. Adaptivity and multiple implementations imply late binding. A program will be more adaptive if it can delay some binding decisions and even rebind them at runtime as conditions warrant. If a programmer makes commitments unnecessarily early, and later the environment does not match what the programmer assumed, then the program will perform poorly.

In this step publications PUBL-01, PUBL-02, PUBL-03 and presentations PRES-04, PRES-05, PRES-07, PRES-08, PRES-09, PRES-10, PRES-11, PRES-12, PRES-13 were produced. This was a key part of this effort, because the early discussions of adding QoS to CORBA were focused on supporting only multimedia and only across a LAN or thereabouts. Unfortunately, any standards efforts which assumed this would be not very useful across a WAN, in our opinion.

1.3 Architecting QuO: a QoS Framework for CORBA (SOW 4.1.3)

In this step we developed the architecture for QuO to meet the requirements developed in Section 1.2 above; the QuO architecture is also described in some of the publications and presentations listed there.

1.4 Experimentation with QuO and COTS (SOW 4.1.4)

In this step we performed experiments on COTS and GOTS CORBA software, developed a QuO visualization environment, and implemented a proof-of-concept prototype of the QuO contract described in Section 2.6 below. The experimentation identified a limitation in Iona's Orbix version 1.3 which made tracing invocations (to collect usage statistics to feed into QuO and its users) impossible. We suggested an enhancement, adding an ID field to the CORBA::Request class, which was agreed to by Iona and will be incorporated into an early point release after Orbix 2.0.

2. Technical Overview

2.1 Introduction

The development and deployment of distributed programs has become increasingly commonplace. Much of this has been made possible by the judicious use of *middleware*, a layer of software above the communication substrate which offers a consistent, higher-level abstraction throughout the network. One increasingly important category of distributed applications is multimedia applications, including video on demand. Such applications demand high-performance communication substrates. Simultaneously, these substrates are offering new features such as quality of service (QoS) and multicast, which these applications could exploit [LMT93, PP92, Topo90, ZDE+93]. For example, QoS allows reservations with guaranteed *system properties*, operational attributes such as throughput and delay which involve particular system conditions. However, these QoS features are offered at the communication substrate level, and much work is needed to enable middleware to translate these features to the application level.

Other important kinds of applications can benefit from QoS support in middleware. For example, an increasingly significant kind of distributed applications is collaborative planning applications [BSB+95, FCO95]. These applications feature widely-dispersed people collaborating using a shared workspace, a video conference, and expert systems to develop a course of action. These applications are created out of many subcomponents which are integrated with CORBA, a middleware standard for distributed object computing [OMG95]. The interactions between these subcomponents feature a much wider spectrum of usage patterns and QoS requirements than typical multimedia applications. In the process of fielding many such applications worldwide, we have observed that these systems have great difficulty adapting to volatile system conditions. They also have difficulty evolving over time to be deployed in different environments, e.g. to be developed on a LAN and migrate to a WAN. The root cause of these problems is middleware's lack of support for handling system properties such as QoS.

CORBA's interface description language (IDL) is an important base for developing distributed applications. IDL describes the *functional interface* to the object, the type signature of the operations which the object embodies, independently of the underlying programming language, operating system, and communication medium. Specifying only the functional interface allows distributed applications to be developed rapidly without regard to the underlying services. CORBA thus uses the traditional model of an interface that hides the implementation details. However, distributed applications based on CORBA's IDL operate acceptably as long as resources are plentiful. For example, experience has shown that CORBA works well where objects are either local (in the client's address space) or within the same LAN as the client, because the system properties of these environments are stable, well understood, and resources are plentiful.

Unlike a LAN, however, in wide-area distributed environments, system properties are more dynamic and hostile, and also more likely to change from configuration to configuration. In order

to field a distributed application over a wide-area network, the usage patterns, the QoS requirements, and the underlying resources must be dealt with. Unfortunately, these features are precisely what is being hidden behind the functional interface described by IDL. Thus, to make a distributed application perform adequately and be evolvable, we need to specify more of the details of the design decisions embodied in an implementation of an object. Also, the implementation must be “opened up” to give access to the system properties of the CORBA ORB and objects. This enables an inexpensive and easy way to alter the implementation without sacrificing the software engineering gains from object-oriented techniques.

Another problem is current application programmers are not normally trained to deal with system properties, such as those embodied in QoS, because the system properties of local objects are ideal. For example, local objects do not fail independently of a client once they are created, for all practical purposes, and the delay is minimal when invoking a method. Further, because of the simple resource model, programming languages mix together the functionality of the code and the optimization of resources. However, the system properties of distributed objects are far from ideal: they can fail unexpectedly, and the delay for a method invocation to return may be long and have high variance. As a result, most invocations to remote objects in a typical distributed application are bracketed with extra code to handle errors and performance conditions.

This section describes issues which must be addressed to support QoS at the CORBA object layer. These issues include integrating system knowledge which is available at different times, from different sources, and in different locations. It also includes reducing the variance of the system properties as seen by the distributed application layer. Finally, it includes exposing key design choices in the implementation of CORBA objects to enable distributed applications to better adapt and evolve.

With these issues in mind, we have developed a framework, Quality of Service for Objects (QuO), to support QoS at the CORBA layer. This framework extends the CORBA functional Interface Description Language (IDL) with a QoS Description Language (QDL). QDL specifies an application’s expected usage patterns and QoS requirements for a connection to an object. The QoS and usage specifications are at the object level (e.g., methods per second) and not at the communication level (e.g., bits per second). An application can have many connections to the same object, each with different system properties. QDL allows the object designer to specify QoS regions, which represent the status of the QoS agreement for an object connection. The application can adapt to changing resources and partial failures by changing its behavior based on the QoS regions of its object connections. Finally, QuO provides many hooks for measuring and enforcing QoS agreements and for dispatching handlers when the agreements are violated.

2.2 Observations about Distributed Applications

Software developers face great difficulties when developing and deploying distributed applications, particularly when they must operate over a wide area. In this section, we describe one important kind of distributed application, collaborative planning systems. We then discuss some of the problems encountered when fielding these and other kinds of distributed

applications. We conclude this section by outlining the solution QuO provides for these problems.

2.2.1 Overview of Collaborative Planning Applications

Collaborative planning applications are an increasingly important category of distributed applications. Such planning could involve military logisticians planning the movement of supplies, military commanders planning an air strike, a manufacturer and its suppliers devising a delivery schedule, or a physician and a specialist jointly analyzing an X-ray. Other examples with a similarly broad spectrum of client-object interactions include a video customer support link or a remote video teller.

Collaborative planning applications can be very complex, featuring dozens of people collectively performing many different tasks. As an example, the structure of the different kinds of interactions between just two of the participants in a typical collaborative planning application is given in simplified form in Figure 1. Here, two users are collaborating at different levels involving a video conference, a shared workspace, and scheduling algorithms. The video conference is used to exchange verbal communication, including synchronizing the workspace. The domain-independent shared workspace allows the scheduling algorithms to construct domain-specific graphical objects, and the collaborators to view, manipulate, and annotate these objects. The domain-specific or even application-specific scheduling algorithms plan a given scenario and display their results on the shared workspace.

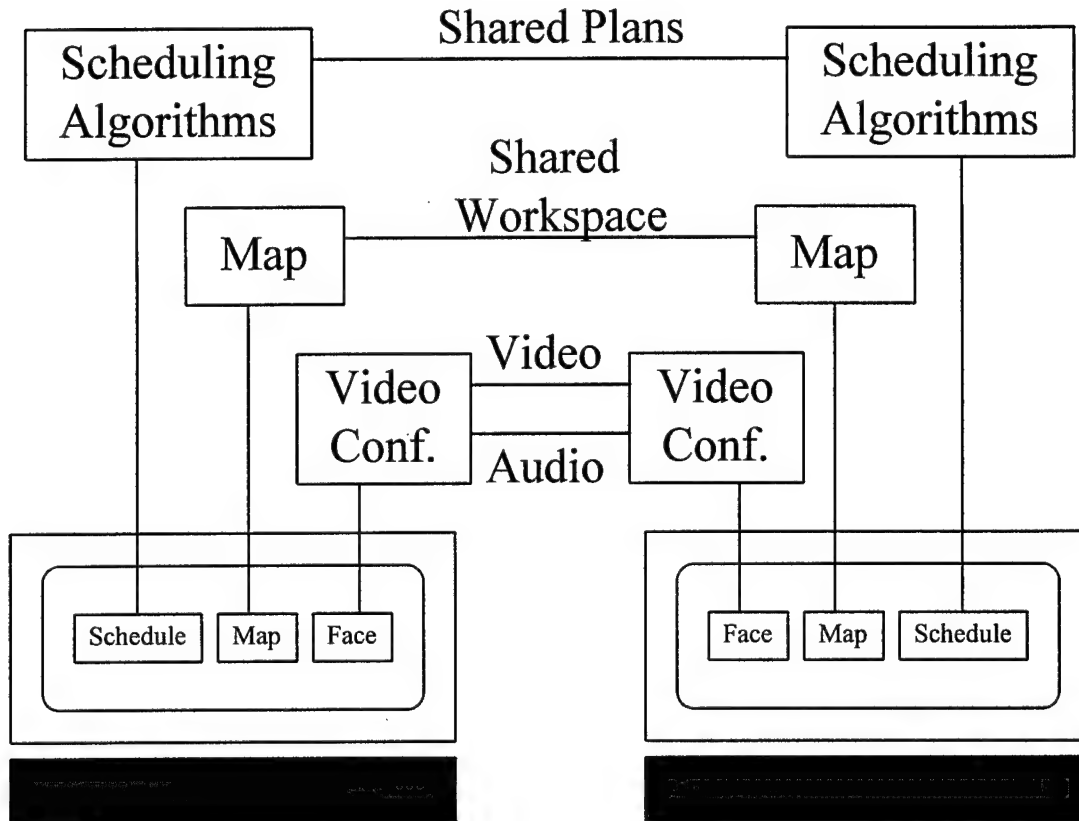


Figure 1: Collaborative Planning Usage Patterns

Each level in this example has its own client usage patterns and adaptivity requirements. One extreme, the usage patterns of the video conference are knowable *a priori* but little variance can be tolerated if the program is to satisfy the user's expectations. The other extreme involves scheduling interactions, whose usage patterns are difficult or impossible to obtain ahead of time but whose delivery allows for significant variance. In between these two extremes of usage patterns is the shared workspace. These three levels also have different abilities to adapt to changing resources while satisfying the user's expectations. The audio portion of the video conference cannot tolerate a lower bandwidth or higher variance to be useful, while the users can generally tolerate the video portion being sent at a much lower rate or even dropped if resource become more scarce. Updates to the shared workspace can suffer additional delays under adverse conditions, but if the delay gets too long then the users will consider the updates untimely and the program's usefulness may be sharply curtailed. The scheduling algorithms can endure much longer delays in worst-case scenarios, because users often will not be waiting for immediate feedback.

2.2.2 Problems in Developing and Deploying Distributed Applications

We have observed a number of problems from deploying collaborative planning and other distributed applications [BSB+95,FCO95,Bien95]. Distributed applications tend to be fragile, e.g. they initially perform poorly and unpredictably when deployed; they are hard to move from the development environment to the field environment or to a different field environment; and components cannot be reused because they are tailored to a particular environment. This fragility is present largely because distributed applications are designed using functional interfaces, i.e. ones which ignore system properties such as throughput, delay, and availability. As a result, programmers either ignore system properties altogether or handle them in an *ad hoc* manner with code fragments scattered throughout the program.

Programmers have difficulty handling the system properties which distributed applications face because they are used to programming assuming the system properties of local objects. Thus, they can generally ignore system properties. Interactions with local objects are typically programmed assuming that they offer infinite bandwidth, no delay, and no variance. This systemic shortcoming has two facets. First, the absolute values of the system properties of local objects are better than that of remote ones. This problem can be overcome in special cases where the programmers have *a priori* knowledge of the worse system properties which their code will have to operate despite. For example, programming for satellite communications is an example of an environment with difficult system properties (low bandwidth, high delay, and frequent failures), but one for which code can be programmed without great difficulty. Second, programmers have great difficulty writing code which can operate over a wide dynamic range of system properties, i.e. variance. As a result, distributed applications can not evolve well to new environments. For example, applications developed for a LAN usually perform poorly in a satellite environment because they were not designed to handle the poor system properties encountered there. Conversely, the same application developed for a satellite environment would also not perform well in a LAN because it would contain superfluous specialized code to compensate for poor system properties.

A compounding factor is that the knowledge of system properties is available at different times, in different locations, and from different sources. Programmers make commitments assuming system properties at various times, ranging from design time to run time. The knowledge of system properties is available at different locations in a distributed system, from the client to the object to the communications substrate, and resources connecting them. This knowledge of system properties is produced by different participants, including the object designer, the client designer, the operations staff, and the end user of the distributed application. If these implicit commitments do not match the actual environment in which an object is used, then the application will perform poorly. For example, the object designer commits to an algorithm to implement a method at object design time based on the assumed environment in which the object will operate. However, the application may be deployed in an environment in which another algorithm to implement the object's method would have been a much better choice.

In summary, distributed applications today are not able to adapt well to changing system properties and resource availability when deployed, and they cannot evolve nearly as readily or easily as is desirable. What is needed is for middleware to support more than just the functional interface to an object, so that system properties can be treated as first class entities.

2.2.3 QuO Solution

QuO is a framework we have developed to solve the above problems in three different ways. First, it integrates knowledge of system properties over time, space, and source; this is discussed in Section 2.3. In order to provide a clean way of reasoning about the system properties of the interactions between clients and objects, QuO employs the concept of a *connection* between a client and object, an encapsulation including the desired usage patterns and QoS requirements specified in the form of a contract. To help simplify the combinatorial problem of dealing with an n-dimensional QoS space in this contract, QuO supports first-class *QoS regions*, which designate regions of operation for the client-object connection. To help the application adapt to different system conditions, QuO supports multiple behaviors for a given functional interface, each bound to the QoS region for which it is best suited. Finally, QuO supports different *commitment epochs*, explicit times which different information about system properties is available to be bound.

The second way QuO solves these problems is through the reduction of variance in system properties. This is discussed in Section 2.4. Since all the information about system properties is spread over time, space, and source, QuO masks variance in system properties using *layers of delegate objects* in the client's address space. Each delegate layer embodies knowledge of system properties for one participant in the connection: the client; object; or CORBA Object Request Broker (ORB). Finally, system properties are encapsulated into first class objects which we call *system condition objects*.

The third way in which QuO solves these problems is by making the design decisions of an object explicit, and allowing the selection of different implementations in a given situation. This is the topic of Section 2.5. QDL includes three sublanguages: a *contract description language* (CDL) to describe the contract between a client and an object in terms of usage and QoS; an *object description language* (ODL) to expose the object's design decisions; and a *resource description language* (RDL) with which to implement a given object implementation. The preliminary implementation features a frame based open implementation itself, where each delegate object has distinguished slots for system conditions, predicate engines to evaluate the QoS regions, etc. This will enable QuO to evolve over time. Finally, QuO's system condition objects can be used at different levels of granularity, which allows for their aggregation. For example, a single system condition object which measures throughput could be bound to a single method invocation from one client, to any method invocation to a group of objects from multiple clients, or something in-between.

2.3 Integrating Knowledge of System Properties

A distributed program is a complex entity. It is created from multiple components, with many design decisions being made about the construction, combination, and behavior of these components. The information on which these decisions are being made is provided by different providers, at different times, and is being consumed by a number of consumers. Consider a simple example involving a single client and a CORBA object it uses. Binding decisions about the functional properties of the object and its use by the client are being made at a variety of times:

- language design time: ways in which a client may invoke an object (e.g., synchronous or asynchronous); functionality of any language libraries.
- language implementation time: implementation choices for a given language feature. For example, an object's slots might be implemented using arrays if the language implementer assumes they will be largely full, or using a hash table if he or she assumes the slots will mostly be empty.
- object definition time: interface (instance template) supported by object instances.
- link time: bind an interface to a particular programming language.
- run time: bind an interface a client uses to a particular implementation.

The previous example showed the different times at which the functional properties of an object are bound. However, its system properties must also be bound if the client's expected usage pattern is to be met and if the client and object are to be able to adapt to changing system conditions. In doing so, it helps a distributed application to be adaptive if it can defer these binding decisions as late as possible, as discussed in Section 2.3.4. For example, the client must specify its expected usage patterns and the object must indicate whether it believes it can support the proposed usage pattern. The client and the object must both be informed when resources are lost or congestion occurs so that they can change their expectations or behavior. A client must inform the object when its expected usage patterns change. Similarly, an object must inform the client when the usage pattern it can support (QoS it can deliver) changes.

The remainder of this section is organized as follows. First, we discuss how QuO collects the disparate knowledge of system properties into an abstraction we call a connection. Next, we describe how the problem of coping with an n-dimensional QoS space can be greatly reduced by QuO's QoS regions. After this we discuss how a distributed application must deploy multiple implementations (behaviors) of a given CORBA functional interface if it is to adapt to changing system conditions. Finally, this section summarizes the different times which QuO provides to enable binding decisions to be made given the knowledge of system properties available at the given time.

2.3.1 Connections

As noted earlier, different system information is bound at different times by different parties. For example, in a traditional client/server architecture, system information is present in three independent places which are difficult to reconcile: the client, the network, and the object. This tripartite division is driven by functional layering, because the communication layer is a convenient interface between the client and the server.

However, if any kind of QoS is to be provided, then this system information must be reconciled, and the question is the best place in which to do this. World Wide Web (WWW) browsers feature the client's functionality moving towards the passive object, with intelligent browsers managing system properties by prefetching, parallel image retrieval, and caching. Another way to reconcile system information is to have the communication network do it with an external management information base (MIB), an approach employed by the QoSockets package and the QuAL language [FY94]. We will take the third alternative by extending the object's knowledge of the system conditions and of its implementation into the client's address space. We accomplish this by the use of delegate objects to implement an abstraction of a client-object *connection* with QoS.

Connections define a boundary where expected usage pattern and QoS requirements between the client and the objects can be agreed upon. All interaction between the client and the objects pass through the connection. Effectively, the boundary of the remote object is moved into the clients address space by using a delegate object. The client will create these local delegate objects, which will in turn bind to the remote object. The delegate objects supports the functional interface of the remote object, and they forward the client's invocations on to it. Having the connection boundary reside in the client's address space is beneficial for two major reasons. First, there is essentially no delay or congestion between the client and the connection object. This is a major advantage; for example, with an external MIB there is essentially an additional layer of QoS between the client and the MIB, and the MIB's job is further complicated since it has to account for this layer. The second advantage is, for all practical purposes, the delegate object will not fail independently of the client. It can, however, monitor the availability of the remote object and of any replicas it has. These two advantages combined allow the connection's knowledge of both the remote object's implementation and the communication substrate which connects them to be exploited to provide for better support for QoS and adaptivity.

Employing a connection is a heavyweight solution, when compared to a local procedure call, but its costs are negligible compared to a remote procedure call. The systems interface to the connections and their delegate objects are described in Section 2.5, and its implementation is described in Section 2.6.

2.3.2 QoS Regions

As noted previously, the varied required system information is bound at different times, in different places, and by different parties. Also, the system conditions will change over time. As a result of this, the usage the client actually generates and the QoS the object actually provides may diverge from their expectations.

QuO handles this divergence by allowing the specification of two levels of system conditions of interest, involving both the client and the object. A *negotiated region* is a named region defined in terms of both client usage and object QoS based on the system conditions in which they will try to operate within. A negotiated region is thus defined in terms of the *expectations* which both the client and the object (via its delegate connection object) set. A typical QuO object will support a number of negotiated regions. Within a given negotiated region there may be many *reality regions*, which are named regions defined in terms of the client usage and object QoS *measured* by the QuO runtime system.

QuO allows for the specification of handler routines to be invoked in either the client or the object when transitions occur between either negotiated or reality regions. A handler for a reality region transition informs the client or connection object when measured conditions change sufficiently; e.g., when its observed behavior is not meeting its expectations. This allows the client or connection object to either take compensatory action to try to operate within its expectations, or to change those expectations. A negotiated region handler informs the client or object when the negotiated region has changed. Since this is a fairly heavyweight operation, possibly involving reallocation of lower level resources, QuO applications will normally try to adapt using reality region handlers.

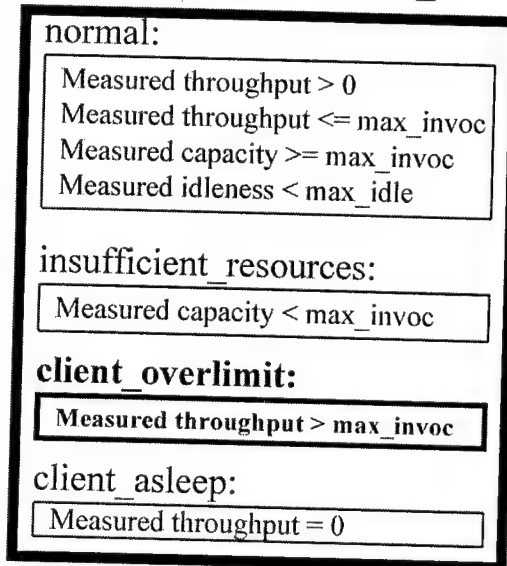
The negotiated (reality) regions may overlap, i.e., the predicates involving the expected (measured) usage and QoS can overlap between negotiated (reality) regions. QuO allows for the specification of a precedence among the regions, which is used to select the current region if the predicate for more than one is true.

Allocated:

Expected throughput > 0

Expected throughput $\leq \text{max_invoc}$

Expected capacity $\geq \text{max_invoc}$



Free:

Expected throughput = 0

Expected capacity = 0

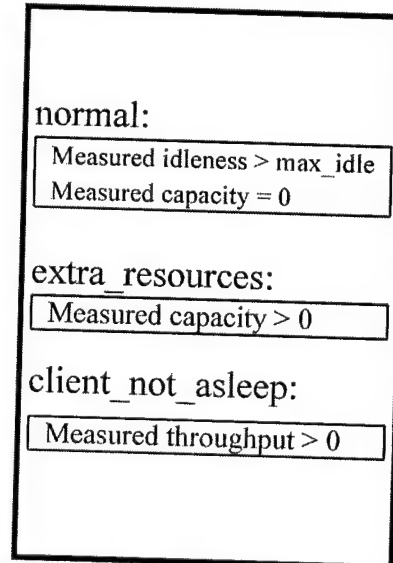


Figure 2: ScreenSaver Regions

Figure 2 shows an example QuO application's negotiated and reality regions. In this application, communication resources are scarce so they must be freed if not in use. This QuO contract, named ScreenSaver, supports two negotiated regions, Allocated and Free. The former is for the normal mode of operation, while the latter is for a mode where the client will be inactive for some period of time. The Free region could be entered because either the client explicitly set its expected throughput to zero (e.g., to indicate its user is going to lunch) or by the QuO runtime system observing no traffic from the client (after *max_idle* seconds). While in the Free Negotiated region, the state and resources associated with the remote object and the connection object will remain allocated, but the resources which were previously allocated to support the delivery of invocations to the remote object may be recycled.

The Allocated negotiated region is where the client expects to generate a throughput greater than zero but not greater than *max_invoc*.¹ The Allocated region has four reality regions: normal, insufficient_resources, client_overlimit, and client_asleep. The normal reality region is where measured throughput and capacity are consistent with the client's and the object's expectations. The insufficient_resources reality region is when the measured throughput and capacity are consistent with the client's and object's expectations. In this case, a handler in the connection

¹ These and other numbers are in units which are specified along with the region's predicates. For example, they may be method invocations per second.

object would be invoked to inform it that its capacity was insufficient. The `client_overlimit` negotiated region is when the client is generating more than its expected throughput; in this case the client will be informed that it is exceeding its expected usage. Reality region `client_asleep` denotes when the client is observed to have generated no throughput for some time (as measured by an idleness detector — it makes no sense to try to measure a throughput of zero!). In this case, a client callback will set its expected throughput to zero, and an object callback will set its expected capacity to zero. This change in expectations triggers a reevaluation of the negotiated region, which will change to Free.

Negotiated region Free is operative where the client expects to generate no throughput and the client expects to support no capacity. Within this region there are three reality regions: normal, `extra_resources`, and `client_not_sleeping`. Reality region normal is in effect when the client's and object's observed behavior is consistent with their expectations. Reality region `extra_resources` is in effect when the object is offering more capacity than expected; in this case, a handler will inform it that it is wasting resources on a dormant client. Reality region `client_not_sleeping` is for when the client has generated throughput when its expectations are zero. In this case, handlers will be invoked to either warn the client or to reset both the client's throughput expectations and the object's capacity expectations. The latter would of course trigger a reevaluation of the expectations predicates, which in this case would cause the Allocated negotiated region coming into effect.

2.3.3 Adaptivity Implies Multiple Behaviors

As noted above, distributed applications need to adapt to changing system properties. The ways in which they can adapt are:

1. Finish later than expected:
 - a. Reschedule for later, when system conditions are (presumably) better, or
 - b. Tolerate finishing later than originally expected.
2. Do less than expected (e.g., compute a less precise answer).
3. Use an alternate mechanism with different system properties. For example, if bandwidth becomes more scarce than computing resources, compress data sent over the network.

QuO provides mechanisms to accomplish these adaptivity schemes. A client and object (via its connection object) may be notified that the reply for an outstanding request will not return to the client with the expected QoS (e.g., within the expected delay). In some cases, the client may be warned that, based on the recent history, any requests in the near future are unlikely to be serviced with the expected QoS. In either case, both the client and the object need to be notified so they can agree upon which of the above options to choose. Also, in the last case, the client

and object need to agree at runtime to change their behaviors. Thus, to allow distributed applications to adapt to changing system conditions, the object designer and application programmer must be able to deploy multiple implementations of a given functional unit. It also needs to have a way to specify which implementations are valid, or at least best, under which system conditions, and be able to dispatch the best implementation at runtime for the current conditions.

QuO provides mechanisms to support all these options. The object and client can be warned of a pending request whose expected QoS is not being met by a reality region callback. They can be made aware that a given QoS is unlikely by the negotiated region which they are presently in. And they can change behaviors in two different ways. The first is accomplished by explicitly calling different internal functions in different negotiated or reality regions. The second is by the object designer specifying alternate paths through the connection object and the remote object by an *object description language*; e.g. to transparently compress the data in the connection object and transparently uncompress it at the remote object, before the invocation reached the target method's body. The different implementation paths are shown in Section 2.4, and the object description language is outlined in Section 2.5.

2.3.4 Summary: QuO Commitment Epochs

In summary, QuO supports object-oriented QoS in a way which allows the application to adapt to changing resources. It allows the object designer and client programmer to defer binding decisions as late as possible. QuO thus supports a number of binding times, which we call *commitment epochs*. They are:

- Definition: define the type of connection using QDL. Also define the structure of QuO regions, as well as hooks to bind handlers to regions to effect different behaviors. Finally, define alternate paths through the object's components (including marshalling and unmarshalling code) for the object, via the ODL.
- Connection: create an instance of the connection object, passing in parameters which bind the shape of the structure defined in definition time (e.g., *max_invoc* above). Bind to the remote object via a particular communication substrate.
- Negotiation: agree upon expectations for client's traffic and object's QoS, i.e., choose the expected bounds to which the client and object will attempt to operate within. In other words, decide upon the agreed behavior.
- Invocation: measure actual client usage generated, object QoS delivered, and failures; i.e., observe actual behavior.

QuO integrates the information about assumed and actual system properties provided at these times and by the various parties to provide the object-oriented QoS it provides.

2.4 Reducing the Variance in System Properties

Programs written using local objects — ones in the client's address space — can assume a much simpler model of system behavior than those using remote objects [WW+94]. For example, the delay in delivering an invocation to a method's implementation and its reply back to the client is negligible, and the throughput (invocations per second) is high. Thus, the absolute values of the system properties are much better for local objects than for remote ones.

The variance of these system properties is also negligible in the local case. While it is possible for a client's host to get overloaded, such occurrences are relatively rare now given the proliferation of inexpensive and powerful workstations and personal computers. In short, resources on the client side are usually much more abundant than on the server side (the remote object) or in the communications link between them. But not only is the absolute value of the system properties higher for remote objects, the variance of these system properties is also much higher than for local objects. Additionally, local objects do not fail independently of their clients, for all practical purposes, while remote ones can. This is another (extreme) form of variance in system properties. Indeed, this higher variance of the system properties of remote objects is generally harder for programmers to deal with than their worse absolute values. For example, if there were little variance, programmers could easily deal with higher delays, lower throughput, etc.; in fact their programs might be structured much like those using local objects. However, programmers using remote objects are forced to include substantial error handling code to deal with the great variance in the system properties.

A goal of QuO is therefore to not only improve the absolute value of a remote objects system properties, as observed by the client, but also to reduce its variance. It accomplishes this through the use of negotiated and reality regions and also by layering internal delegate and system condition objects. QuO applications can adapt to some variance by the use of reality regions, which bind different client behaviors (implementations) to different system conditions. This allows the client and object to keep the connection within its negotiated region of operation as long as possible, thus reducing variance. Further, QuO employs multiple layered delegate objects on the client side, and each layer is able to both mask out some variance of system conditions as well as improve their absolute values by employing the algorithms (implementations) most appropriate for the current conditions.

The remainder of this section is organized as follows. Section 2.4.1 gives an overview of how QuO's delegate objects help mask out variance in system properties. Section 2.4.2 describes how the system knowledge from the various producers is integrated by the QuO architecture to accomplish this, and gives further details of the masking of system properties. Finally, Section 2.4.3 describes why system properties are first class objects in QuO.

2.4.1 Masking Variance with Layers of Delegates

Improving system conditions cannot be accomplished in one time and place, since the systems knowledge which is required to do this is available at different times and in different places. Also, knowledge of how best to use this information is spread among multiple partners. We

layer the delegate objects in the client's address space with this in mind, and an example is given in Figure 3, and greater details are presented in Section 2.4.2. In this figure, the delegates each use their knowledge of the system information they possess to improve the system conditions seen by the layer above them by masking. Each layer exports a negotiated region to the layer above. It uses various techniques (changing policies, etc.) to mask any changing conditions and maintain the QoS it provides to the layer above. When it cannot maintain the QoS corresponding to the current negotiated region, it propagates this information upward via a handler indicating a change in reality region. Each party tries to adapt (by changing policies, etc.) and if it cannot, it indicates a change in expectations. This triggers a renegotiation of the negotiated region, because negotiated regions are defined in terms of the expectations of the client and object. The changing conditions which can be handled by this architecture include changing in resource availability in the network or on a host (which of course affects the QoS which the object can deliver), change in a client's usage pattern, and the failure of an object.

For example, in the figure, the client knows it can (or must) wait an indefinite (or at least long) period of time for an invocation to the object to complete, even if the object fails and must be restarted, or if it has multiple replicas. This is indicated in QDL, and the QDL compiler generates a client delegate object for the application to use. The application invokes the functional methods on the client delegate, and the delegate passes this invocation down and handles exceptional conditions resulting from this invocation. For example, if the remote object or the network link to it fails, the client delegate will indicate that the invocation has been interrupted and will attempt to reestablish the link or create another instance of the object. Once this has been completed, it will pass the invocation down again.

The client delegate is optional, and in many cases will not be used. It is available, however, for those clients who do wish to simplify their invocation semantics in a particular way; e.g., to eliminate much application-level exception handling.

The object delegate can take advantage of the knowledge of its different implementations to adapt to changing conditions while still providing a useful QoS to the client. For example, in the figure, the object delegate will automatically use compression if the bandwidth the ORB is delivering is insufficient to meet the client's needs; it will trade off additional CPU cycles to maintain a desired bandwidth. In many cases this can be done transparently to the client. However, while the bandwidth of a compressed region may be the same as a normal one, other system properties such as the delay may be different, so the object delegate indicates it no longer expects to be able to meet the conditions for the normal region but can for the compressed region. Section 2.5 discusses how open implementation techniques allow the object designer to act on this information to enable the layers, particularly the object, to use the implementation of that object most appropriate to the current conditions.

The ORB delegate translates from CORBA system exceptions and other system information available from the ORB into QuO negotiated regions. In doing so, it can take advantage of knowledge it has of the ORB's implementation, since there will be one ORB delegate class created for each different ORB which QuO supports.

The client, object, and ORB delegate objects are automatically generated by the QDL compiler. At connection time the client passes in references to objects to handle the callbacks (e.g., for a change in reality region) as specified in the QDL. It can optionally pass in one or more system condition objects to be used by the delegate objects as specified in the QDL, e.g., for aggregation. If the client passes in no condition object reference then the QuO runtime system will create one at connection time.

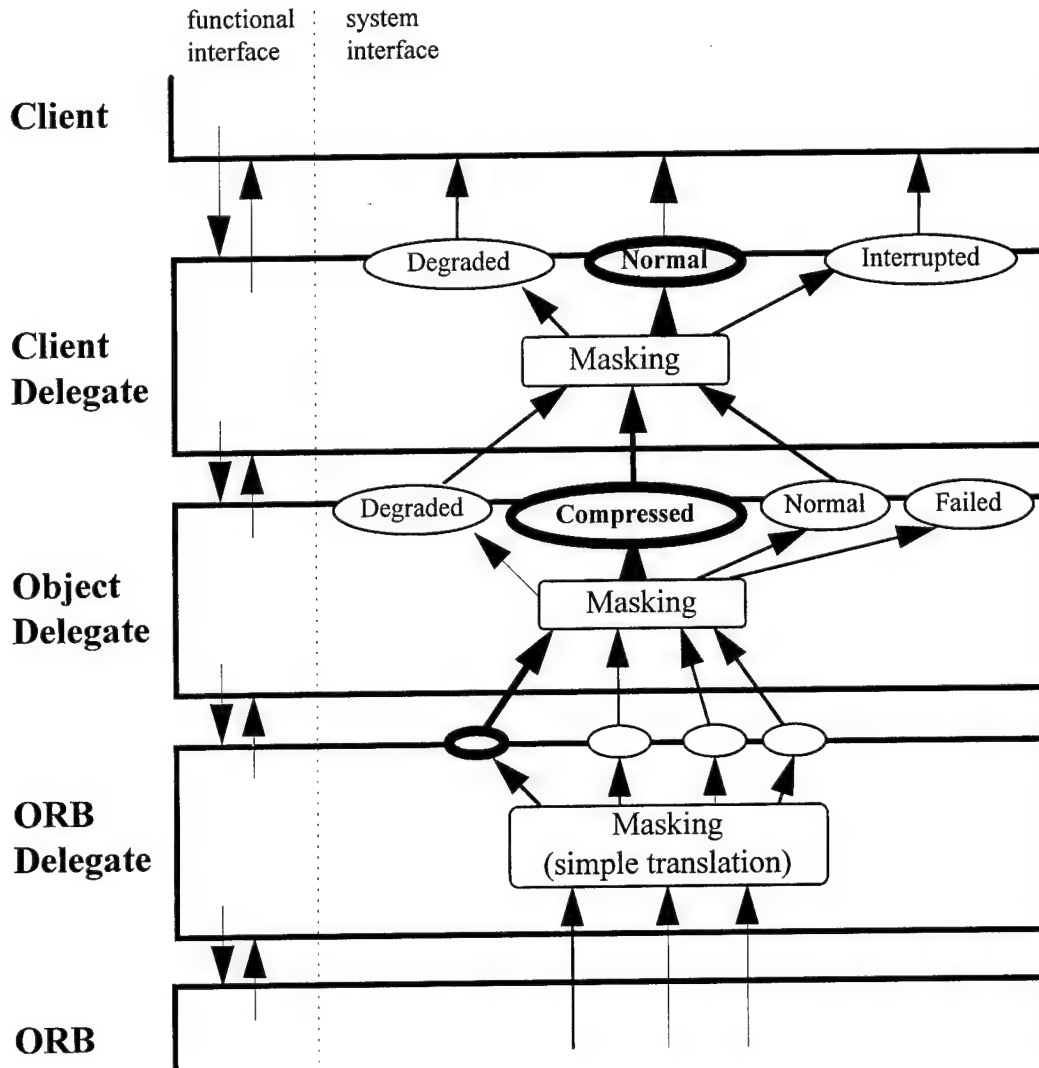


Figure 3: Masking System Properties

2.4.2 Integrating System Knowledge from Different Sources

Figure 4 shows how the QuO architecture integrates knowledge from the different parties into one cohesive framework. In this example, a functional interface named *inv* is being extended to

include system properties with a QDL connection named *invScreenSaver*. This stack is set up when the client connects to the remote object. The participants with different systems knowledge are:

- Client designer: Knowledge of how to adapt to changing reality and negotiated regions, given the different usage patterns it may generate as well as the different implementations with which it can implement its part of the application program. This knowledge is encapsulated in the client delegate object.
- Object designer: knowledge of its different implementations, as encapsulated in the object delegate object.
- QuO designer: knowledge of how the ORB which QuO has been ported to is implemented, as encapsulated into the ORB delegate object.
- ORB designer: a very simple model of system conditions, encapsulated into CORBA system exceptions.
- Operations staff: knowledge of resource availability, resource access permissions, administrative domains, etc., encapsulated in the environment delegate objects.

The delegate objects support different standardized interfaces to facilitate the integration of the knowledge from these varied sources. Each delegate object in the inside of the stack implements various interfaces. This includes the functional interface, so that a client's invocations can be passed down towards the remote object. It also includes a callback interface based on the delegate object below it, for use by reality and negotiated region callbacks, as well as an expectations interface which the lower delegate object can use to query the expectations of the layer above it. It implements a negotiated region interface for use by the delegate object above it; this interface indicates the current negotiated region of the delegate object. Finally, it implements an environment callback interface so it can be informed of changing environmental conditions.

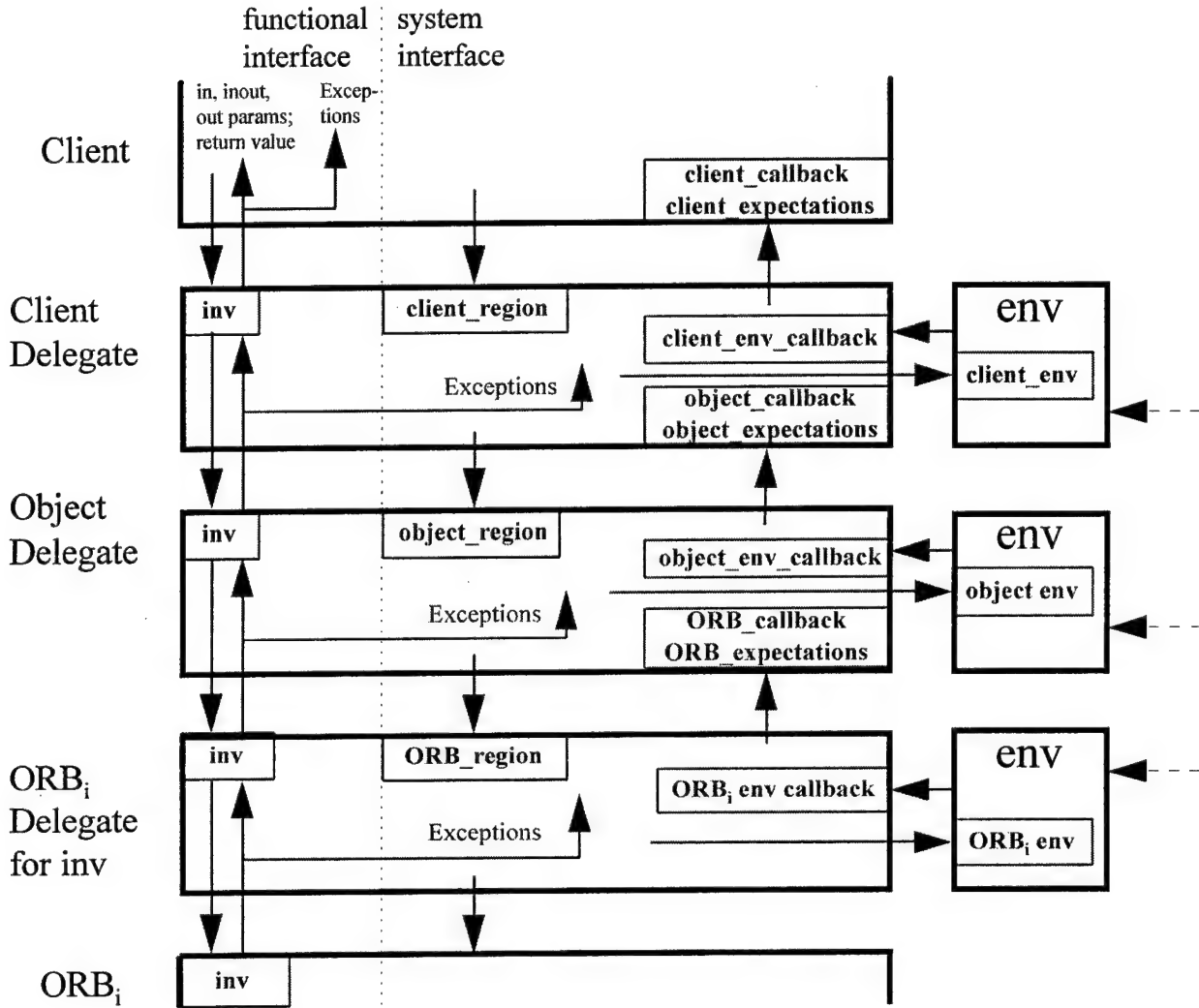


Figure 4: Delegate Object Layers

2.4.3 System Conditions as First Class Objects

In the preceding discussion, note that delegate objects are translators. They translate from lower-level system conditions with higher variance (and often worse absolute values) into more desirable system conditions with lower variance (and often better absolute values), i.e. system properties closer to that of local objects. In this scheme, there are two flows of information: functional and system. The functional flow passes on the client's functional invocation, and also updates the appropriate system condition concerning this invocation (e.g., its pre- and post-

methods log this invocation). The system information propagates independently of the functional information.

To facilitate the flowing of this system information, we encapsulate the knowledge of a particular system condition into a separate system condition object. An example of this is shown in Figure 5. Here the measurement object tracks a client's usage patterns. The connection object can invoke it to register a client invocation or to compare its present value to that of a reality region. The client can test the value of the measurement object if it wishes.

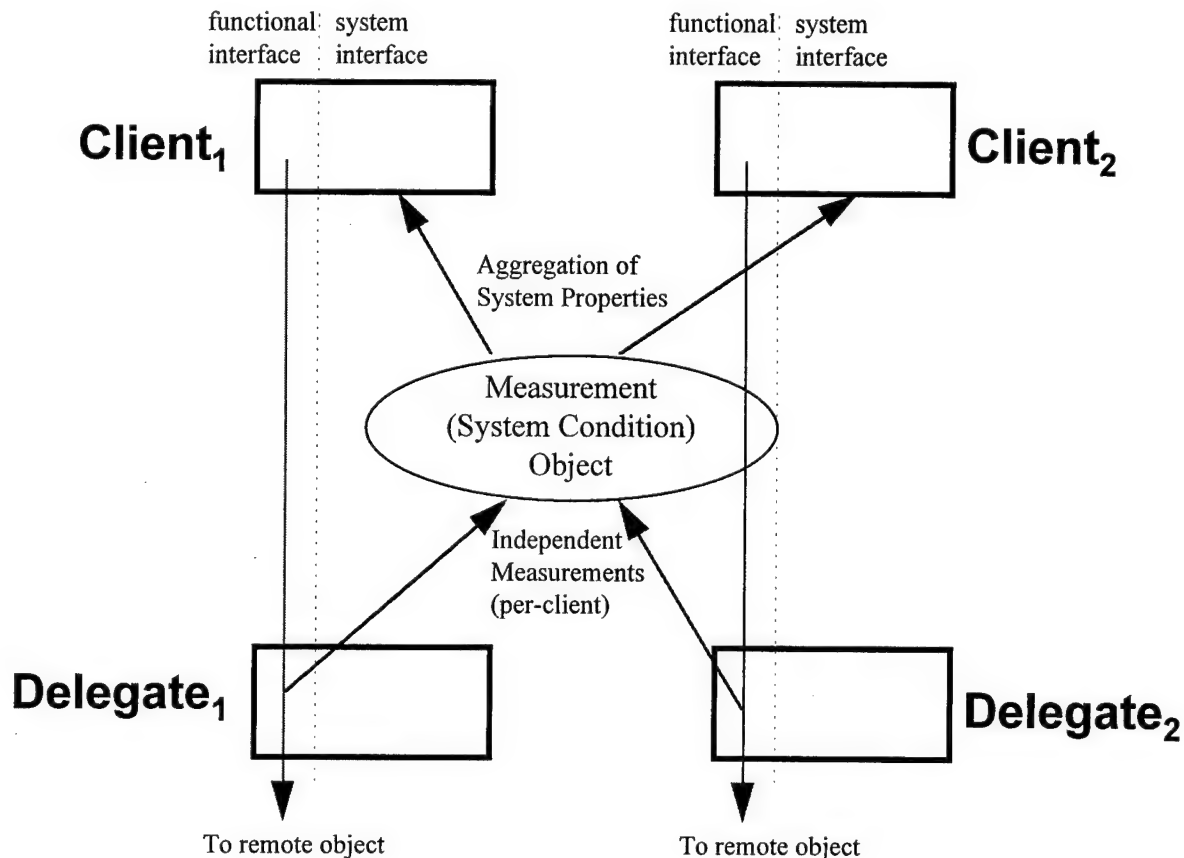


Figure 5: Aggregation via a Shared System Condition Object

This example shows an additional benefit of independent condition objects: they facilitate sharing. For example, multiple clients can share the same measurement object, so that invocations by either are credited against the same measurement object. Since measurement objects will be closely integrated with reservations, this allows the aggregation of multiple connections over the same ATM virtual circuit, for example.

2.4.4 Summary

QuO employs layers of delegate objects to help reduce the variance of the system conditions of remote objects. This allows the client to deal with system conditions which are much closer to that of local objects, both in absolute terms and in their variance. This is very beneficial, since the system conditions of local objects are much easier to deal with and are much more familiar to programmers. Finally, system condition objects encapsulate a system condition on which clients and objects will base a decision as how to adapt or behave.

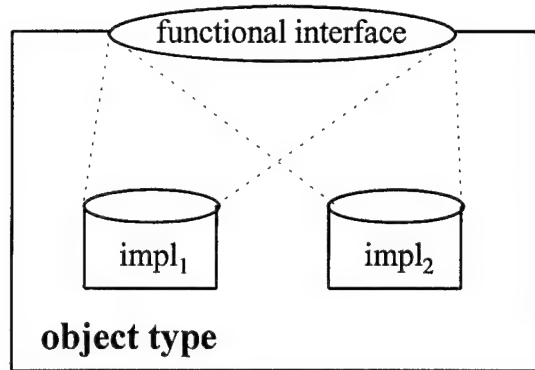
2.5 Open Implementation Techniques and QuO Objects

Large distributed systems developed by the DoD and by others have grown to staggering levels of complexity over the last few decades. To manage this complexity, today's large systems are built using layers of abstraction. Each layer provides services to the client layers above it. Each abstraction provides an interface which provides the service but hides the details of the underlying implementation. While a new implementation of an abstraction can replace an old one, there is typically only one implementation of that abstraction operating at any one time.

Unfortunately, layered abstraction based only on a functional interface is not enough. A single implementation of a component cannot provide everything needed by all possible applications. The fundamental reason for this is because the most appropriate implementation design for a "black box" module cannot be determined without knowledge of how the client intends to use the module.

The inadequacy of this approach has lead to *ad hoc* workarounds, either through "programming behind the lines" or by simply rewriting large parts of the module in a manner more suited to their needs. For example, programmers are often very aware of the target hardware and the way their programming language orders loops when writing nested loops to access a multi-dimensional array. Indeed, High Performance FORTRAN (HPF) allows the programmer to annotate array declarations with layout specifications which control how the arrays are distributed across processors.

In the last few years a new discipline, *open implementation* (OI), has emerged to help augment the black box functional interface to allow the designer of an object to expose key design decisions which affect the object's performance and reliability. An open implementation provides disciplined, object oriented access to the implementation of the abstraction. This allows the application developer to alter the behavior of the application by choosing the implementation of a component best suited to the application. A *metaobject protocol* (MOP) allows the client to provide an object to choose the implementation for a module which best suits its needs, as shown in Figure 6. Here the application programmer writes a small metaobject to choose the implementation of an object with the system properties which most closely match its intended operating environment.



metaobject protocol:

```

if property > threshold then use impl1
else                               use impl2

```

Figure 6: A MetaObject Protocol

Information on which a MOP can make its decision is available both at compile time, through global code analysis, and at runtime, through values provided by the middleware. Runtime MOPs (RTMOPs) are MOPs which can use runtime values to make this choice, while compile time MOPs (CTMOPs) can only use information available at compile time.

Open-implementation is an important approach for the development and deployment of distributed systems because:

- Meta-level aspects are specified separately from the functional aspects of a system, so they can be changed more easily.
- A service using OI is more general because it is architected to be able to handle a wider range of system properties.
- A meta architecture allows a system to reason about itself and change its behavior based on both the knowledge of itself and of the current system properties.

The QuO architecture must support exposing two things:

1. The functional interface of an object: how a given functional interface is implemented.
2. Knowledge of how system properties propagate (our version of how system information is propagated). This allows us to stack delegate objects to successfully reduce the variance of system conditions.

In exposing this information, QDL must be able to describe:

- The expected behavior of the interface: the CDL includes the expectations and the reality of invocations to the object.
- The automatic deployment of alternate implementations of the object, based on the expected usage patterns provided by the CDL. The description is done with object description language (ODL). The range of system conditions an implementation is suitable for is described in terms of the expectations and reality given above, and the implementation's underlying requirements are specified in terms of abstract resources (processing, storage, communications, etc.).
- The actual resources required to implement a given object implementation. This is accomplished with a resource description language (RDL), which maps the above abstract resources into real ones on the current host. The QuO runtime system can use the knowledge of the current system properties, which alternate implementations are more appropriate for the current system properties (from the ODL), and which resources are available to those implementations for executing on.

2.6 Implementation

We have developed a proof-of-concept prototype implementation of the ScreenSaver contract shown in Figure 2. It works with two ORBs, BBN's Corbus (CORBA-compliant Cronus) and Iona's Orbix v2.0 ORB, demonstrating QuO's portability across ORBs. The C++ objects which implement the regions of the contract are shown, and the CDL for the ScreenSaver contract is given in Appendix B.

2.6.1 C++ Objects to Implement the CDL

The C++ objects which implement the contract for ScreenSaver are given in Figure 7. The client calls a connection manager to set up the contract for it, passing in *max_invoc* and *max_idle* and any parameters to identify which remote object of the given interface to connect to (e.g., Orbix's arguments for *_bind*). The client is returned a pointer to the Object Delegate, which it uses to make its functional invocations to the remote object. Functional invocations also generate sidecalls from the Object Delegate to its Contract object (for ScreenSaver), which record the invocation and perform any other system properties synchronous with an invocation.

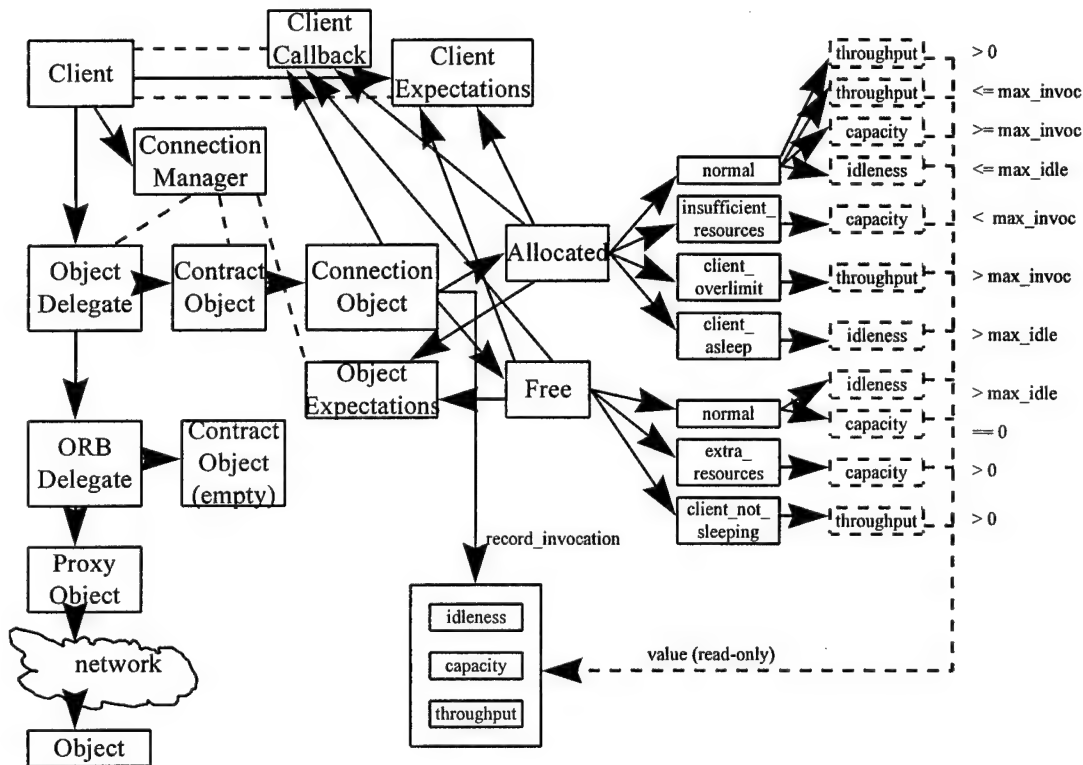


Figure 7: C++ Objects which implement the ScreenSaver contract

2.7 Related Work

A few efforts have recently begun to bring QoS to the CORBA layer. The ODP Trader architecture provides a means to offer a service and to discover a service which has been offered [OMG-ODP]. It does have the concept of a “property” which can be used to specify non-functional characteristics of the service being offered. However, the properties are not defined but rather are general placeholders. Further, the Trader architecture seems to be intended not for direct implementation but as a general architecture on which to create more detailed standards with which to implement.

One such specialization is the TINA effort [LGF95]. TINA is an ongoing effort by telecommunications providers and computer vendors to enable the rapid deployment of telecommunication services, with a focus on real-time multimedia applications. It does involve a contract specifying both the client’s usage and the object’s QoS provided. However, the set of system properties on which the contract is based is limited. Further, it is not adaptive. QuO permits the specification of multiple regions of operation for a contract, both negotiated and reality regions. QuO provides this because we long ago realized that in wide-area distributed applications contracts will inevitably be broken, and it is important to provide an orderly way for the application to handle this. TINA has no such provisions for adapting to the breaking of a

contract. While this is probably a very good assumption and a proper optimization for a video server delivering a movie a few miles over ATM, it is a poor assumption over the wide area.

The open implementation idea is an outgrowth of research in meta-object protocols (MOPs), which is part of the Common Lisp Object System (CLOS). CLOS is an object oriented language implemented in terms of meta-objects and protocols between them. A programmer can extend the behavior of the language by providing his own meta-classes. The MOP approach is now being applied in more general contexts [KL95,MMA+95,Kic91,Kic94,Meta95,And95a,And95b,LS94]. Application areas include fault tolerance [FNT95], distributed objects [Chi95,CM93], and operating systems [KL93].

2.8 References

- [And95a] Kenneth R. Anderson, Freeing the Essence of a Computation, ACM Lisp Pointers, VIII, 2, 1995. [ftp://openmap.bbn.com/pub/kanderson/fater95/essence/essence.ps](http://openmap.bbn.com/pub/kanderson/fater95/essence/essence.ps)
- [And95b] Kenneth R. Anderson, Compiling a Metaobject Protocol, in Preparation. To be submitted to Reflection '96.
- [Chi95] Shigeru Chiba, A Metaobject Protocol for C++, OOPSLA '95, p285-299.
- [Bien95] Bienkowski M. Demonstrating The Operation Feasibility of New Technologies. *IEEE Expert*, February 1995, pp 27-33.
- [BSB+95] Burstein, Mark H. and Schantz, Richard E. and Bienkowski, Marie A. and desJardins, Marie E. and Smith, Stephen F. The common prototyping environment. *IEEE Expert*, February 1995, 17-26.
- [Chi95] Shigeru Chiba, A Metaobject Protocol for C++, OOPSLA '95, p285-299.
- [CM93] Chiba, Shigeru and Masuda, Takashi, Designing an Extensible Distributed Language with a meta-level architecture, ECOOP '93.
- [FCO95] Fowler N. and Cross S. and Owens C. The Arpa-Rome Knowledge-based Planning and Scheduling Initiative. *IEEE Expert*, February 1995, 4-9.
- [FNT95] Fabre, Jean-Charles, Nicomette, Vincent, and Perennou Tanguy, Implementing fault tolerant applications using reflective object-oriented programming. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, IEEE, Pasadena, California, June 1995, p.489-498.
- [FY94] Florissi, P. and Yemini, Y. Management of application quality of service. In *Proceedings of the Fifth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1994.
- [Kicz91] Kiczales, G. and J. des Riveres and D. G. Bobrow. The Art of the Metaobject Protocol, The MIT Press, 1991.

- [Kicz92] Kiczales, Gregor, Towards a new model of abstraction in the engineering of Software, IMSA '92 Proceedings (Workshop on Reflection and Meta-level Architectures).
- [Kicz94] Kiczales, G. ed. Workshop on Open Implementation '94, Internet Publication (URL <http://www.parc.xerox.com/PARC/spl/eca/oi/workshop-94>).
- [KL93] Gregor Kiczales and John Lamping, Operating Systems: Why object-oriented", Proceedings of IWOOS '93.
- [LGF95] Leydekkers, Peter and Gay, Valerie and Franken, Leonard. A computational and engineering view on open distributed real-time multimedia exchange, Proceedings of NOSSDAV '95, Boston, USA, April 1995.
- [LMT93] Leslie I. and McAuley D. and Tennenhouse, D. ATM Everywhere? *IEEE Networks*, March 1993.
- [LS94] V.B. Lortz and K.G. Shin, Combining Contracts and Exemplar-based Programming for Class Hiding and Customization, OOPSLA '94, p. 453-567.
- [LZ95] Arthur H. Lee and Joseph L. Zachary, Reflections on Metaprogramming, *IEEE Transactions on Software Engineering*, 21:11, November, 1995, 883-893.
- [Meta95] Proceedings of the Workshop on Advances in Metaobject Protocols and Reflection (Meta '95), part of Ninth European Conference on Object-Oriented Programming (ECOOP '95).
- [MMA+95] Hidehiko Masahura and Satoshi Matsuoka and Kenichi Asai and Akinori Yonezawa, Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation, OOPSLA '95, 300-315.
- [OMG95] Object Management Group. *The Common Object Request Broker: Architecture and SpecificationI, 2.0 (draft) ed.*, May 1995.
- [PP92] Partridge C, and Pink S. An implementation of the Revised Internet Stream Protocol (ST-2). *Internetworking: Research and Experience*, March 1992, 27-54
- [Topo90] Topolcic C. An experimental Internet Stream Protocol: Version 2 (ST-II). *Internet RFC 1190*, October 1990.
- [WW+94] Waldo, Jim and Wyant, Geoff and Wollrath, Ann and Kendall, Sam. A Note on Distributed Computing. *Report SMLI TR-94-29*, Sun Microsystems Laboratories, November 1994.
- [ZDE+93] Zhang L, Deering S, Estrin D, Shenker S, Zappala D. RSVP: a New Resource ReSerVation

3. Conclusions and Recommendations

3.1 CORBA as a Wide-Area Research and Development Vehicle

Distributed object computing is an excellent vehicle for conducting research into issues involved with wide-area middleware as well as for deploying applications across it. To see this, consider network operating systems. They typically offer three kinds of resources: communications, storage, and processing. However, these resources are not integrated in any fashion, which makes their combined use much more difficult than if they were integrated.

Technology	Encapsulates + Integrates		
	comm.	processing	storage
Dist. Objects: CORBA (or net. OLE)	Yes	Yes	Yes
sockets	Yes		
SQL		Weak	Yes
RPC	Yes	Yes	
Distributed File Systems	Yes		Yes
Message- Oriented Middleware	Yes		Weak
RSVP, ATM, ST-2	Yes		

Figure 8: Middleware and Reservations Resources

Figure 8 summarizes how different kinds of middleware and communications reservation substrates encapsulate these three categories of resources:

- **Distributed objects** not only encapsulate all three kinds of resources but also integrate them in a very powerful and simplified manner. CORBA is an industry standard for distributed object computing with a flexible but powerful specification as well as a rich array of object-level common services and application-level common facilities. There are many commercially available CORBA distributed processing products on the market now. In addition, there exists a government-owned ORB, Corbus, which is freely available for advanced government R&D projects. A network version of Microsoft's OLE is under development and its first

release should occur soon. We feel that CORBA is a much better base for government research and development than network OLE, and this will remain true in the foreseeable future. CORBA's object technology is arguably 3-4 years ahead of network OLE's, and OLE does not support inheritance, a key object-oriented feature. Further, CORBA is an open standard created by the OMG, the world's largest consortium consisting of over 500 software and hardware vendors, end users, and government agencies. CORBA products are available from multiple vendors. On the other hand, network OLE will only be available from Microsoft, and this sole source constraint should be a concern for organizations (such as the DoD and other government agencies) who need to develop distributed systems which must be maintained over long periods of time and by many suppliers.

- **Sockets** encapsulate a communications path (e.g., a TCP/IP connection) and the resources required to maintain it (e.g., buffers) but do not include processing and storage abstractions (the buffers are not an independent storage abstraction but are just used to implement the communications).
- **Structured Query Language (SQL)** is a standard for relational databases. It encapsulates storage by separating the logical organization of the data from its physical implementation, but provides only weak processing abstractions in the form of a database query. SQL proper does not address distribution and hence communications. Also, the distributed object model is more general than a relational database and indeed can subsume it, which is one important thread of current CORBA standards efforts.
- **Remote Procedure Call (RPC)** makes an invocation to a procedure body on a remote host appear to the programmer similar to how a local invocation would. (Its runtime appearance may vary greatly from a local procedure call, however, due to failures and network latency) RPC thus encapsulates communication and processing but not storage.
- **Distributed File Systems** such as Sun's Network File System (NFS) make files available across a network, and thus encapsulate communications and storage but not processing.
- **Message Oriented Middleware (MOM)** provides shared queues for producers to deposit messages to and consumers to withdraw them from. These queues can be distributed or even replicated or persistent. MOM thus encapsulates communications but offers only a limited storage abstraction in its message queues, but does not encapsulate processing.
- **RSVP, ATM, and ST-2** are communications substrates which offer reservation of communications. However, they do not reserve the processing and storage necessary to provide end-to-end QoS for distributed objects.

3.2 DCNTN Summary

The DCNTN project created a framework for providing application-level adaptivity for CORBA applications. While originally conceived as an effort to factor new developments in communication technology into evolving middleware concepts, the DCNTN project evolved into conceiving additional layers of software to address the non-functional aspects of providing an effective distributed computing environment. In addition, QuO serves as an organizing framework for uniformly tying together a number of previously distinct system properties, integrated through the object metaphor.

3.2.1 Technical Conclusions

Among the conclusions we have made during the course of this activity are the following:

- System issues dominate the task of creating large distributed applications
- CORBA solves the **functional** problems of distributed computing, but does not mandate a specific implementation
- Few research issues remain in the area of **functional** integration; however there are still many engineering issues
- CORBA does not address the **systems** issues involved in distributed computing (e.g. performance, availability, security)
- Many systems related research issues remain, and CORBA is the best currently available architectural building block for testing out ideas
- Application-level adaptivity must be provided for applications to be more robust across the wide area environments typical of DoD systems, because at the application level the best end-to-end requirements and tradeoffs can be made. A key to providing this application-level adaptivity is runtime binding.
- Some key research questions remaining include:
 - How to characterize the **system requirements** of an application
 - How to characterize the **system properties** of a mechanism
 - How to use the **appropriate mechanism** for a given situation

3.2.2 QuO Summary

During the design and prototype implementation phases of the project we developed the concept of Quality of Service for Objects (QuO). The major accomplishment of the QuO design is in adding quality of service concepts to the CORBA layer in a manner which

- makes system properties first class entities in an object's interface,
- makes explicit the assumptions about the system properties under which an implementation of an object is appropriate,
- simplifies the N-dimensional QoS space into a small number of application defined regions of operation with different operating characteristics,
- allows an application to adapt to changing system conditions through the mechanisms of compound contracts and multiple implementations.

3.3 Limitations of the QuO Approach

QuO is a technology which has the promise to help DoD applications adapt much better to the hostile WAN environments in which they must be deployed. However, limitations and risk factors include:

- At this early stage QuO must be considered speculative and in need of further development and evaluation with actual deployed DoD applications to validate and improve on its technology.
- QuO requires more from the application programmer than would just ignoring the non-functional issues. However, programs which do this do not work for long across a wide area,

and QuO seems to be much simpler technology to use than if the programmer had to try to handle all these non-functional issues by hand.

- Widespread use of QuO assumes the availability of COTS software for distributed object programming. These are not yet available from a wide variety of vendors (as C++ compilers are, for example), and there is always the possibility that CORBA could be overtaken (mainly for non-technical reasons) by network OLE.

3.4 Future Directions

This project has only begun to scratch the surface of the very important area of handling systems issues to help applications distributed across a wide area be more adaptable at runtime and evolvable at design time. As a result of this activity, we recommend the following next steps:

- Integrate and test with appropriate communications substrate offering wide area reservations and high performance operation; e.g., RSVP.
- Integrate other system properties beyond network performance into the quality of service paradigm to evaluate its adequacy as an organizing tool for integrating and trading off various system properties in the construction of large distributed application systems.
- Continue development of the contract languages: enhance CDL and develop ODL and RDL.
- Tools to help design CDL graphically, visualize runtime activity of QuO-based applications, trace and visualize client-object invocations to help ascertain an application's requirements, etc.

The status of the QuO implementation, as well as short-term and long-term implementation directions, is given in Figure 9.

	Now	Soon	Later
CDL Compiler	•Hard-coded contract library (ScreenSaver)	•Generate contract objs for any contract	
Delegate Generator	•Generates functional delegate methods for any CORBA-2 IDL		•Maybe use a PD IDL parser instead of Corbus gencppclient
Connection Manager	•Hard-coded for this particular delegate + contract stack	•Will create single obj. contract for any IDL and contract	•Set up stacks/graphs with many contracts •Handle shared system condition objects •Reconnect connection
System Condition Library	•Recursive throughput •Idleness detector (one per process limit)	•Leaky-token thr'put •Unlimited # idleness detectors •Delay	•Capacity with network management system •Fault tolerance •Security
Language	•C++		•Java, maybe Ada95
ORBs	•Corbus 2.0 •Orbix 2.0		•Neo •Post Modern?
Misc.			•RDL+ODL •Invocation Tracing

Figure 9: QuO Implementation Status and Future Directions

Currently the QuO implementation is a proof-of-concept prototype designed to demonstrate the feasibility of the technology. It does not have a CDL compiler but rather uses a hard-coded contract (the ScreenSaver contract). Its delegate generator was based on the Corbus gencppclient and will generate functional delegates for any CORBA-2 IDL. The connection manager is hardcoded for this particular contract and delegate stack. The system condition library now includes a recursive throughput object, a mock capacity management object, and an idleness detector object. (This present implementation only supports one idleness detector object instantiation per process.) A C++ client API is supported viz. the CORBA-2 IDL to C++ mapping for the object delegate, client expectations, and client callback; the entire collection of delegate and contract objects are also implemented with C++. Finally, two CORBA ORBs are supported: BBN's Corbus version 2.0 and Iona's Orbix version 2.0, both CORBA-2 compliant ORBs.

In the very near future we plan to offer a simple but general implementation of QuO with arbitrary contracts. To do this, we will build a simple CDL compiler to generate the contract

objects for any CDL contract. The connection manager will then be extended to manage any combination of functional IDL and contract as specified in the CDL. The system condition library will be extended to include the leaky token algorithm for throughput measurement, the end-to-end delay of an invocation, and by removing the one-per-process limitation of the current idleness detector implementation.

In the long-term much more work is needed to develop QuO's potential to facilitate wide-area distributed object development and deployment. The delegate generator would probably be more maintainable if it were separated from the Corbus gencppclient program and instead used a public domain IDL parser. The connection manager should be extended to set up more general stacks/graphs with nested contracts, handle shared system condition objects (e.g., a throughput measurement object shared between two connections), and to re-establish a connection when it fails. The system condition library should be augmented to take network capacity (and other status) information from an external network management system such as the Communications Anchor Desk currently under development by BBN. Also, other kinds of system conditions besides network performance should be supported, including fault tolerance and security. This is useful not only to provide these system conditions to the application, but also to help understand both the application-level tradeoffs which are useful and common as well as the design and implementation tradeoffs involved in implementing multiple kinds of system conditions. In the future more ORBs should be supported. The resource description language (RDL) and object description language (ODL) will be developed and implemented. Finally, an invocation tracing package should be provided to allow application developers and deployers to be able to ascertain their invocation patterns (something virtually none of them know anything about now) so that they can better specify the application's requirements with CDL.

Appendix A: Publications Presentations, and Other Participation

A.1 Publications

[PUBL-01] "Supporting Quality of Service for CORBA Objects", John A. Zinky and David E. Bakken and Richard E. Schantz, Theory and Practice of Object Systems (special issue on the OMG and CORBA), to appear in January 1997.

[PUBL-02] "QoS Issues for Wide-Area CORBA-Based Object Systems", David E. Bakken and Richard E. Schantz and John A. Zinky, Proceedings of the Second International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 96), IEEE, Laguna Beach, CA, Feb. 1-2, 1996, to appear.

[PUBL-03] "Overview of Quality of Service for Distributed Objects", John A. Zinky and David E. Bakken and Richard E. Schantz, Proceedings of the Fifth IEEE Dual Use Technologies and Applications Conference, Utica, NY, May 1995.

[PUBL-04] "Distributed Computing over New Technology Networks Interim Summary: Quality of Service for Distributed Objects", BBN Systems & Technologies Report 8076, May 1995 (9 month Interim TR).

[PUBL-05] "Distributed Computing over New Technology Networks Final Report: Quality of Service for Distributed Objects", BBN Systems & Technologies, May 1996 (this document).

A.2 Presentations

A.2.1 Rome Lab Presentations

[PRES-01] DCNTN Kickoff Meeting, September 1994.

[PRES-02] "Distributed Computing over New Technology Networks", Richard E. Schantz Rome Lab/C3AB 17th Technology Exchange Meeting (TEM-17), October 1994.

[PRES-03] DCNTN 6 Month Review, February 1995.

[PRES-04] "Distributed Computing over New Technology Networks", Richard E. Schantz Rome Lab/C3AB 18th Technology Exchange Meeting (TEM-18), November 1995.

[PRES-05] DCNTN 12 Month Review, November 1995.

[PRES-06] DCNTN Final Review, May 1996.

A.2.2 Other Presentations

[PRES-07] "Quality of Service for CORBA Objects: Distributed Computing over New Technology Networks (DCNTN) Project", John A. Zinky, IETF Integrated Services Working Group, Danvers, MA, April 1995.

[PRES-08] "Quality of Service for CORBA Objects" John A. Zinky, ACM SIGCOMM 95 Middleware Workshop on Distributed Objects and Procedures, August 1995.

[PRES-09] "Quality of Service for CORBA Objects", John A. Zinky and David E. Bakken and Richard E. Schantz, invited talk at GTE Laboratories, Waltham, MA, June 1995

[PRES-10] "Overview of Quality of Service for Distributed Objects", David E. Bakken, Fifth IEEE Dual Use Technologies and Applications Conference, Utica, NY, May 1995.

[PRES-11] "QoS Issues for Wide-Area CORBA-Based Object Systems", (Panel presentation), David E. Bakken, Second International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 96), IEEE, Laguna Beach, CA, Feb. 1, 1996.

[PRES-12] "Quality of Service for CORBA Objects", David E. Bakken, Invited lecture, U. Mass Lowell, graduate course on distributed objects, GTE Labs campus, Waltham, MA April 4 1996.

[PRES-13] "Research Issues in Quality of Service for CORBA", John A. Zinky, invited research talk (focus on wide-area systems with large number of objects), MIT Artificial Intelligence Lab, April 30, 1996.

A.3 Other Participation

[PART-01] Attendance at Twenty-Fifth International Symposium on Fault-Tolerant Computing, David E. Bakken, Pasadena, CA, June 1995.

[PART-02] Demonstration Talk and of QuO Visualization Environment at Rome Lab/C3AB 18th Technology Meeting (TEM-18), John A. Zinky and David E. Bakken, November 1995.

[PART-03] Attendance at OMG Meeting, David E. Bakken, San Jose, CA, June 1995.

[PART-04] Attendance at Rome Lab/C3AB 17th Technology Exchange Meeting (TEM-17), James C. Berets and John A. Zinky and David E. Bakken, October 1994.

[PART-05] Attendance at Rome Lab/C3AB 18th Technology Exchange Meeting (TEM-18), John A. Zinky and David E. Bakken, November 1995.

Appendix B: Contract Description Language (CDL) for ScreenSaver Contract

```
// Forward declarations for classes used in the module's parameters. The first 3
// are required, since the client needs to pass in a client_callback object reference
// and needs to be returned a negotiated_region reference and a
// client_expectations reference.
interface ScreenSaver_client_callback;
interface ScreenSaver_negotiated_region;
interface ScreenSaver_client_expectations;

connection invScreenSaver(

    // Parameters required for every QDL connection
    in ScreenSaver_client_callback cl_call,           // for client_callback
    in ScreenSaver_client_expectations cl_exp,       // for client_expectations
    out ScreenSaver_object_expectations ob_exp,      // for object_expectations

    // Parameters specific to this connection. Can be used in predicates for
    // negotiated and reality regions. These must all be in arguments and
    // also (for now) be IDL basic types such as float, long, etc.
    in double max_invoc,
    in double max_idle
) is

    // IDL-based interfaces for required interfaces. An object reference for each one
    // will be maintained by the connection object, and may be used by the meta-level
    // parts of the connection, notably the contract

    // The functional_base is the interface for the object which this connection is
    // extending (adding QoS to). The connection object will create
    // a reference to the remote object
    // when the client creates the contract, and the connection object will pass back
    // this reference to the client when it is finished initializing itself.
    functional_base interface inv
```

// The client_callback is the interface which the contract will use to notify the client
// of any transitions between negotiated regions or reality regions.
// The client will pass in a reference to an object supporting this when the
// connection is created.

client_callback interface ScreenSaver_client_callback

// The object_callback is the interface which the contract will use to notify
// the connection object (which can notify the remote object
// if desired) of any transitions between
// negotiated regions or reality regions. The connection object will create an object
// supporting this interface when the **connection**/contract is being initialized

object_callback interface ScreenSaver_object_callback

// The client_expectations is the interface for the client to set its expectation
// variables. These will be used in the contract, along with the object's
// expectations, in the predicates which define each negotiated region.

client_expectations interface ScreenSaver_client_expectations

// The object_expectations is the interface for the object to set its expectation
// variables. These will be used in the contract, along with the client's
// expectations, in the predicates which define each negotiated region.

object_expectations interface ScreenSaver_object_expectations

separate reality regions for ScreenSaver::Allocated:

normal:

```
when QuO_condition.measured_throughput > 0 m_p_s and
when QuO_condition.measured_throughput <= max_invoc m_p_s and
when QuO_condition.measured_capacity >= max_invoc m_p_s and
when QuO_condition.measured_idleness <= max_idle secs
```

insufficient_resources:

```
when QuO_condition.measured_capacity < max_invoc
```

client_overlimit:

```
when QuO_condition.measured_throughput > max_invoc m_p_s
```

client_asleep:

```
when QuO_condition.measured_idleness > max_idle sec
```

// precedences tell which reality regions are chosen if more than

// one predicate is true

precedence normal, client_asleep, client_overlimit, no_resources

// Invoke methods in either the client's or the connection/

// object's callbacks when certain reality region

// transitions are made in in negotiated region Allocated.

transitions callbacks are

normal -> insufficient_resources:

```
// Warn the client that there isn't enough capacity, even though we're in
// negotiated region Allocated and thus there is supposed to be capacity.
```

```
client_callback->warn_no_resources()
```

```
// Tell the object or connection to get busier.
```

```
object_callback->allocate_capacity(max_invoc)
```

insufficient_resources -> normal:

```
// Let the client know that it doesn't have to hold its breath any more
```

```
client_callback->warn_enough_resources()
```

any -> client_overlimit:

// Let the client know it is exceeding its negotiated promise
client_callback->warn_overlimit(max_invoc)

any -> client_asleep:

// Let both the object and the client know that the client has gone asleep.
// One or both may reset their expectations (e.g., the client's throughput
// or the object's capacity), which could cause a renegotiation.

client_callback -> warn_sleeping()

object_callback -> client_asleep()

end transition callbacks

end separate reality regions ScreenSaver::Allocated

separate reality regions ScreenSaver::Free:

normal: **when QuO_condition.measured_idleness > max_idle sec and**
 when QuO_condition.measured_capacity == 0 m_p_s

extra_resources: **when QuO_condition.measured_capacity > 0 m_p_s**

client_not_sleeping: **when QuO_condition.measured_throughput > 0 m_p_s**

// Invoke methods in either the client's or the object delegate when certain reality
// region transitions are made in in negotiated region Free
transitions callbacks are

any -> extra_resources:

object_callback->deallocate_capacity()

any -> client_not_sleeping:

client_callback->get_to_sleep()

// Instead of telling the client to get back to sleep, we could have a
// different policy which tells object to allocate the capacity again:
// object_callback->allocate_capacity(max_invoc)

end transition callbacks

end reality regions ScreenSaver::Free:

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.